Report No. 3182

INTERFACE MESSAGE PROCESSORS FOR THE ARPA COMPUTER NETWORK

QUARTERLY TECHNICAL REPORT No. 3

1 July 1975 to 30 September 1975

Principal Investigator: Mr. Frank E. Heart
Telephone (617) 491-1850, Ext. 470

Sponsored by:
Advanced Research Projects Agency
ARPA Order No. 2351, Amendment 15
Program Element Codes 62301E, 62706E, 62708E

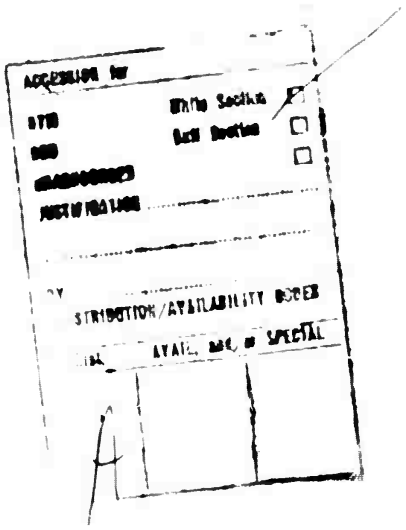
Contract No. F08606-75-C-0032
Effective Date: 1 January 1975
Expiration Date: 30 June 1976
Contract Amount: $2,384,745

Title of Work: Operation and Maintenance of the ARPANET

Submitted to:

IMP Program Manager
Range Measurements Lab.
Building 981
Patrick Air Force Base
Cocoa Beach, Florida 32925

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.


White Section
Buff Section
STRIBUTION/AVAILABILITY CODES
AVAIL. and/or SPECIAL

(14) BBN Report No. 3182

Bolt Beranek and Newman Inc.

(11) October 1975

(12) 32 p.

(6) INTERFACE MESSAGE PROCESSORS FOR THE ARPA COMPUTER NETWORK.

(9) QUARTERLY TECHNICAL REPORT NO. 3,
1 July - 30 September 1975,

(10) Frank E. Heart

(15) F08606-75-C-0032,
ARPA Order No. 2351

Submitted to:

IMP Program Manager
Range Measurements Lab
Building 981
Patrick Air Force Base
Cocoa Beach, Florida 32925

This research was supported by the Advanced Research Projects Agency of the Department of Defense and monitored by the Range Measurements Laboratory under Contract No. F08606-75-C-0032.

060 100

TABLE OF CONTENTS

1. OVERVIEW

This Quarterly Technical Report, Number 3, describes aspects of our work on the ARPA Computer Network under Contract No. F08606-75-C-0032 during the third quarter of 1975. (Work performed in 1973 and 1974 under Contract No. F08606-75-C-0027 has been reported in an earlier series of Quarterly Technical Reports, numbered 1-8; and work performed from 1969 through 1972 under Contract No. DAHC-69-C-0179 has been reported in a still earlier series of Quarterly Technical Reports, numbered 1-16.)

Several IMPs were delivered during the third quarter. A 316 IMP was installed at Scott Air Force Base. An ARPA-owned 316 Satellite IMP was installed at Goonhilly in the UK and a privately-owned 316 Satellite IMP was installed at Etam, West Virginia; these two Satellite IMPs permit the initiation of multi-access satellite communication experiments. The IMP which was delivered to NSA during the second quarter was installed in the network this quarter. We also delivered the network's first Pluribus IMP, to SDAC, late in the third quarter. This IMP was undergoing hardware and software shakedown at the end of the quarter and was expected to be installed in the network on an operational basis on the first day of the fourth quarter.

In addition to the above, a cyclic relocation of IMPs in the Cambridge, Massachusetts, area was carried out during the quarter. The intent was to permit higher bandwidth at the CCA node. Accordingly, the CCA TIP was moved to MIT, the MIT 316 IMP was moved to Harvard, and the Harvard 516 IMP was moved to CCA.

We have also been active in a number of other areas during the quarter, as discussed below.

Security Activities. During the quarter our security activities centered primarily on the carrying out of PLI TEMPEST tests, refining the PLI hardware and software, and beginning formal PLI documentation. A complete PLI, an IMP, and extensive test equipment were shipped at the beginning of the quarter to NESSEC in Washington, D.C., where the TEMPEST tests were carried out. These tests resulted in a generally satisfactory "bill of health" for the PLI. Taking advantage of the availability of an actual Key Generator (KG) at NESSEC and the insight into the PLI operation gained during TEMPEST tests, we ran a number of additional tests which flushed out some residual hardware and software bugs. We also ran tests to confirm our previous estimates of PLI performance when operating with the KG. These tests have suggested some hardware features which, if feasible, could improve performance. The PLI equipment will be shipped

back to BBN early in the fourth quarter where it will be readied for reshipment to the field. In the documentation area, we began software documentation and have almost finished a functional specification and interfacing manual which will be published as an appendix to BBN Report 1822.

We also participated in an ARPA-organized meeting to begin planning an experimental program to demonstrate the feasibility of securing a packet-switching network while maintaining much more flexibility than is possible with the PLI, and we produced a working document outlining an approach to such flexible packet network security.

Network Control Center Program. During the quarter the Network Control Center program was modified to be able to handle the situation of more than sixty-three IMPs in the network. The program is now assembled to permit about eighty IMPs with a variable number of Hosts (in particular more than four) per IMP, although the basic program modification is such that with only a trivial reassembly the program can handle many more than eighty IMPs. During the process of modification the program was reformatted to permit it to be assembled by our PDP-10 cross assembler rather than our PDP-1 cross assembler, and NCC program development will henceforth be done on our PDP-10. This move

from the PDP-1 to the PDP-10 is in keeping with our general goal of moving NCC tasks off the aging PDP-1. Also, network throughput data collection, which was previously done on the PDP-1 with copies sent to the PDP-10, is now done entirely on the PDP-10.

Network Hardware Maintenance. The third quarter was the first during which BBN provided hardware maintenance directly rather than obtaining it through a subcontract to Honeywell. Our experience this past quarter has confirmed the wisdom of our switch to direct maintenance. In particular, July was the best month in network history from the point of view of hardware maintenance; and the quarterly average was very good as compared with previous quarters.

Terminal IMP. During the quarter we reached a minor milestone with regard to TIP operation and software maintenance. For the first time within memory (possibly ever), all TIPs in the network are now running the same version of the TIP software system. Not content to let things rest, we are quite far advanced in our new Telnet implementation for the TIP, and we are still aiming for new Telnet operation by the beginning of next year. Also, early in the fourth quarter the necessary TIP software change will be started to allow the TIP to be used when

more than sixty-three IMPs are on the network. We also hope this change will be done by the beginning of next year.

316 IMP Development. The 316 IMP program has undergone a period of consolidation this past quarter to clean up and prepare for the massive change which will be implemented in the fourth quarter to permit more than sixty-three IMPs on the network. Prior to entry into the consolidation phase, the IMP program had remained stable for three full months, one of the longest periods without change in network history. In addition to a number of very minor modifications, the consolidation has included reclaiming some storage both for buffers and for future code, implementing a cleaner packet reload system, and implementing a load/unload package (which permits the trace, statistics, and VDH packages to be installed and removed almost at will). We are now prepared for the actual implementation of the more-than-sixty-three-IMP change. During the third quarter we also wrote the modification to BBN Report 1822 to cover the new formats required by the change, and a new edition of this report will be distributed as early as possible in the fourth quarter. Finally, during the third quarter we have essentially completed the development of a version of the Very Distant Host package which can run in extended core of an IMP or TIP so that it does not have to occupy IMP buffer space.

Pluribus IMP. As specified in our contract, our efforts with the Pluribus IMP hardware development had almost ceased as of the beginning of the third quarter. We have continued to spend effort finishing Pluribus documentation to add to the three documents that have already been published. Two more Pluribus documents, numbers 4 and 6, are almost done and should be published early in the fourth quarter. Documents 7 through 9 are being compiled. Document 3 is still some way in the distance. Also, we have spent a little time this quarter developing a version of the Pluribus IMP modem interface which meets V.24 and V.28 standards. In the Pluribus IMP software area, we have continued to improve the program both to keep up with 316 IMP changes and to tidy up a number of loose ends of the Pluribus IMP implementation. Naturally, the actual installation the Pluribus IMP at SDAC has forced a different mode of software development than has previously been used; as with the 316 IMP, the Pluribus IMP software must now continue its development in a style which uses very small incremental changes while always maintaining an operational system.

Satellite IMP. During the quarter the bulk of our efforts in the Satellite IMP area were directed to the installation and beginning of field operation and maintenance of Satellite IMPs at the Etam and Goonhilly earth stations. The Etam Satellite IMP

was installed at the end of July with very little difficulty. The Goonhilly Satellite IMP was installed in August with somewhat more difficulty. Both systems have now been operational together and with the rest of the network since official service began at the beginning of September. Also, we attended a meeting of ARPA, Western Union International (the record carrier for the U.S. side of the satellite link), and COMSAT for the purpose of thrashing out maintenance procedures for the satellite circuit. Finally, we attended a meeting of the group of institutions which will actually carry out the experiments involving the Satellite IMPs, at which planning for the experiment was begun.

2. A METHOD FOR DETECTING INTERRUPT BUGS

Both the IMP and the TIP have required and continue to require development and maintenance of somewhat complicated, interrupt-driven, real time programs. As is typical with such programs, "interrupt bugs" or "multiprogramming bugs" have occasionally crept into the programs. Since such bugs are among the most difficult to diagnose, and since we have (slowly but) almost continuously had to modify our programs, thus providing opportunities for bugs to creep in, we were forced for the sake of self-preservation to develop a systematic method of assuring the absence of multiprogramming bugs.

This systematic method has been in use for over two years and, while not perfect, has in fact caught many such bugs which might otherwise have caused IMP and TIP problems. Consequently, although "methods of automatic program verificaion" are a bit outside the normal scope of our contract, we think the method has been of sufficient direct value to network maintenance to justify the following report.

2.1 The Problem

Multiprogramming bugs are those which can occur when two processes within a system attempt to modify a variable simultaneously or when one process interferes with the control structure of another by incorrectly sharing a subroutine with it. These bugs occur in multiprocessing systems, where the conflicting processes are truly concurrent, as well as in multiprogramming systems, where only one process can be active at a time. A bug of this type is frequently very difficult to locate because neither process can detect any anomaly. Furthermore, since a failure caused by such a bug depends on the coincident timing of the conflicting processes, a failure may occur only once a minute, once an hour, or even less frequently. Also, problems often will not show up until the next time the variable is referenced, which may occur long after and far removed from the actual source of the fault.

Dijkstra and others have done much theoretical work on the general problem of coordinating independent processes. However, when execution speed and program size are critical factors, the general procedures they developed are too expensive to use every time a variable is referenced. Further, in many environments the actual protection techniques available are fairly

straightforward. It is our belief, then, that what is needed is not further work on "automatic" protection schemes, but rather techniques for identifying those portions of a program which actually require protection, thereby allowing the remainder of the program to be left unencumbered. Our work has focussed on these practical problems within the context of assembly language programming for small, real-time systems.

We concentrate for the most part on interrupt-based systems because this is the environment with which we are most familiar and in which we have done the most analysis. Conceptually, however, all multiprogramming systems merely simulate a multiprocessing system. The primary attribute of such systems is the requirement for rapid response to external events. Interrupts are the most common architecture with which such systems are constructed. Although the techniques we have developed are designed for interrupt-driven systems, they apply with little significant modification to the entire spectrum of multiprogramming systems.

As an example of a common interrupt bug, consider a system in which there is a loop to process data and an interrupt routine to service an input device supplying the data. The routines communicate by means of a variable which is incremented by the

interrupt routine each time a datum arrives and is decremented by the processing loop each time a datum is processed. The variable is zero when the system is idle, indicating that there is no data waiting to be processed. Now, let us hypothesize that the interrupt routine and the processing loop look something like:

```
INTERRUPT:    SAVE ACCUMULATOR
              LOAD VARIABLE
              INCREMENT ACCUMULATOR
              STORE INTO VARIABLE
              RESTORE ACCUMULATOR
              EXIT

LOOP:         LOAD VARIABLE
              TEST ACCUMULATOR EQUAL TO ZERO
              IFSO GOTO LOOP
              DECREMENT ACCUMULATOR
              STORE INTO VARIABLE
              PROCESS A DATUM
              GOTO LOOP
```

An observant systems programmer will notice that the above routine, as simple as it is, contains an "interrupt bug". Assume that the variable has some value, n. If a datum arrives just after the LOAD VARIABLE instruction, an interrupt will occur and the interrupt routine will increment the variable to n+1. When the loop is resumed, the now-incorrect value of n will be restored to the accumulator and will get decremented, and instead of containing n at the beginning of the next loop, the variable will contain n-1; a datum will have been lost.

One way to fix this bug is to rewrite the processing loop as follows:

```
LOOP:   DISABLE INTERRUPT SYSTEM
        LOAD VARIABLE
        TEST ACCUMULATOR EQUAL TO ZERO
        IFSO GOTO ENAB
        DECREMENT ACCUMULATOR
        STORE INTO VARIABLE
        ENABLE INTERRUPT SYSTEM
        PROCESS A DATUM (we assume no conflict here)
        GOTO LOOP

ENAB:   ENABLE INTERRUPT SYSTEM
        GOTO LOOP
```

The variable is being shared between two routines, and for a small portion of the time its value is "incorrect": from the LOAD until the STORE. During this period, the interrupt routine must be prevented from running and using the incorrect value of the variable, and the simplest way to do this is just to disable the entire interrupt system, preventing all interrupts.

A fairly simple rule suggests itself: if any routine modifies a shared variable, it must prevent any other routine from using the variable while it is being changed. This rule is the crux of interrupt programming.

2.2 A Systematic Approach to the Problem

The first step in understanding a system is to determine its interrupt structure, which usually follows from an analysis of the desired response characteristics of the system and its I/O devices. This will result in a directed graph in which the nodes correspond to the interrupt routines and the arcs indicate which routines should not be interrupted by which others. For typical computers, the interrupt system will constrain this structure to be a simple linear ordering of routines by priority; in others it is possible to have a more complex structure. (In a linear structure, if B cannot interrupt A and C cannot interrupt B, then C cannot interrupt A. Some computer architectures would permit C to interrupt A unless the software explicitly prohibits it.)

In our particular case, although the computer in fact poses no constraints, we chose to implement a linear system. There are two primary reasons for this choice. First, more complicated structures offer little, if any, advantage over a linear system. Second, the interrelationships of the interrupt routines become much more complicated when structure is non-linear. Except as noted, for the remainder of this report we will deal exclusively with linear interrupt structures. The approaches and analyses can be carried over to more complicated structures fairly easily,

but in doing so the rules we describe will rapidly become very difficult.

The next step is to identify the system's interrupt routines. This identification is usually a simple matter of starting at each interrupt entrance and tracing out the control paths until the interrupt exits are reached. Non-reentrant shared code and subroutines are treated as though they were variables; that is, the techniques described below for dealing with variables are applied to determine an "effective interrupt level" for the code or subroutine, and then the analysis proceeds as though the code were actually a part of the level thus determined, the procedure repeated iteratively until all of the code in the system has been associated with an interrupt.

For each variable in the system, the highest priority routine which references it may do so with impunity. However, all lower routines must take care that they are not interrupted while modifying the variable in question. If the lower routines lock out interrupts at (and implicitly below) the level of the highest routine, they will be assured error-free access to the variable.* Thus, for each variable we must determine an

---

* It is primarily the absence of this property that makes non-linear structures difficult to deal with.

"effective interrupt level", the level at which interrupts must be disabled (either implicitly by being the interrupt routine at that level or explicitly in lower levels) to guarantee safe access. To do this, examine each such element and determine which routines share it. The "highest priority" routine is the one which cannot be interrupted by any of the other routines sharing the element; its level is assigned to the element.

In an existing system, locating an interrupt bug probably requires some degree of insight into the structure of the program. Nonetheless, by methodical application of the above rules, it is possible to verify the structure of an entire system and discover any bugs present in it. In fact, for some bugs this may be the fastest way to locate them.

If we number the interrupt levels so that a given level can be interrupted by all lower-numbered levels and implicitly inhibits all higher-numbered levels, then the effective (or "hardware") levels we have just assigned are simply the lowest numbered levels which use each element. Now, when any higher numbered routine uses an element, it must at least disable interrupts up to that level. Notice that what the routine is actually doing is making itself appear to the interrupt system as

though it were the lower numbered interrupt. Thus, one can view the situation as being that the lower priority routine has, through software, "become" a higher priority routine. We refer to this procedure as making the routine modify its "software" interrupt level.

The hardware level of a routine reflects which routines it can interrupt, and the software level indicates which routines are prevented from interrupting it. At the point of the interrupt entry, these levels are the same, and a routine can neither alter its hardware level nor make its software level higher than its hardware level. It can, however, decrease and restore its software level as necessary to insure safe access to any variables it might need.

Our basic approach consists of a mechanism which simply maintains and displays the hardware and software levels for each line of code and variable in the system. With this information as a guide, the programmer can proceed on his own to resolve any conflicts that are pointed out. The level information is inserted, modified and acted upon in a completely manual fashion; it is less an augmentation of our programming system than it is a documentation technique. Nonetheless, it has proven valuable for several reasons: 1) the rules are quite simple and

deterministic, and they lead to a high level of program correctness when properly applied; 2) the documentation aspect of the system is, in itself, valuable; and 3) forcing the programmer to be aware of multiprogramming issues improves the overall reliability of his programs.

There are six macros to declare the hardware and software levels of the program. In addition to setting the levels, the macros perform various consistency checks. The first three assemble appropriate code to achieve the desired effect upon the interrupt system:

1) INT N declares the interrupt entrance at level N.
2) INH N locks interrupts at level N.
3) ENB restores the interrupt system to the current hardware level.

Because programs have transfers and subroutine calls and non-contiguous fragments of code, there are analogs to INT, INH and ENB which effect the declaration for the purposes of checking but presume that the interrupt system is already at the declared level, and thus do not generate any code:

4) LEV N declares the code to be hardware level N. This is an implicit INT.
5) LCK N declares the code to be at software level N. This is an implicit INH.
6) RET declares the code's software level to be equal to its hardware level. This is an implicit ENB.

Each word of code that is assembled and shown in the listing is accompanied by its hardware level, and also by its software level if different. Variables are indicated with a V and the determination of their effective levels and the verification of their correct use is done manually, or otherwise, distinctly from the assembly process. For example, some levels that have been compiled for our IMP program include:

```
M2I = 0   Modem-to-IMP
I2M = 2   IMP-to-Modem
I2H = 3   IMP-to-Host
H2I = 4   Host-to-IMP
T.O = 5   Timeout
TSK = 6   Task
BCK = 7   Background
```

A sample use of these levels (in TASK) might be:

```
6     LDA THIS        /GET THIS PACKET
6     INH I2M         /LOCK OUT I2M
6 2   STA EMQ XI      /ADD NEW PACKET TO QUEUE
6 2   STA EMQ X       /EMQ = END OF MODEM QUEUE
6 2   ENB             /COME BACK TO TASK LEVEL
6     JMP FOO
```

The effective levels for THIS and EMQ must be"known" (for example, by reference to a previous assembly). THIS is on level 6; it is a temporary in Task. EMQ is on level 2; it is shared by I2M, Timeout and Task.

The INH-ENB mechanism is somewhat simplistic; in practice, things can be quite a bit more complicated. The principles

always remain the same, but it is not always clear just how to prevent a given routine from running. There can be assorted interlocks in the software: it might be that a particular interrupt cannot occur (for example, because the device is known to be inactive), or it might be that the higher priority routine cannot take a path which accesses the variable in question (for example, in setting up the parameters of a datum before a flag indicating that there is a datum has been set). However, once the key places in the system are pointed out, it is almost always easy to implement the controls or verify that they are already present.

In systems with more complicated interrupt structures, applying the rules becomes correspondingly more difficult. Nonetheless, even the most intractable of structures will yield to the steadyhanded application of the rules. We give three examples. First, consider two user-level jobs within a timesharing system. The timesharing system itself is probably a complicated interrupt-driven system, but let us direct our attention only to the two user jobs. The nature of a timesharing system is that at any point a user job can be interrupted, "swapped out", and another user job run in its place. User jobs typically have no explicit way to "inhibit" one another. So if two jobs wish to communicate through some shared structure, be it

shared memory or a shared disk file, considerable care must be given to prevent interrupt bugs from cropping up. Again, the mechanisms chosen will vary widely, but the basic rules can be used again and again to pinpoint those portions of a program which require protection.

Second, consider two processes (for instance, coroutines) that both run at background level, which suspend and resume processing on some basis (this is, in essence, the heart of a polling system). If these two processes share a common resource the following kind of bug can occur:

```
A                          B

LOAD X                     LOAD X
...                        ...
(suspend processing)       (suspend processing)
(resume processing)        (resume processing)
...                        ...
STORE X                    STORE X
```

Because B can run between the time A suspends and resumes and vice versa, either's STORE X could be an error if the other has run and changed X. Again, the problem is that a routine modifies a shared variable in an "interruptable" way. A systematic solution to this problem can also be developed based on the assignment of local levels within the main hardware level to all routines and variables and the use of the rule that a routine implicitly inhibits all lower priority levels and only its own

sublevel -- each sublevel considers the other sublevels at its level as being "higher" than it.

Our third example of a situation requiring an extension to our systematic approach is that of a true multiprocessor. Here the problem of concurrent access to shared resources is present at all times. Protection in such an environment can be effected by the well-known uninterruptable "test and set" instruction. However, now a new type of problem arises: when routines require several "locks" at once in order to proceed, deadlocks can occur if all routines do not take and release locks in a coordinated way. An extension to our approach is of value here: if the locks are assigned levels and routines are constrained never to take a lock with a lower number than one it already has (i.e., locks must be taken in strictly ascending order), our system can be expanded to provide the necessary sequencing information to allow the avoidance of lock-ordering deadlocks.

2.3 Automating the Approach

In this section we discuss a means of making our approach automatic. The procedure we are going to describe is a "cookbook" for assigning levels and verifying that there are no bugs. Much of this work could be done by a program, and the procedure has been organized with that in mind. However, we have not actually implemented this automatic approach.

For the first step, we identify and assign hardware levels to all of the executable code in the system. We start at the actual hardware interrupt entrances and assign to them the actual level of the interrupt they represent. We then trace through the code in the natural way, assigning the starting hardware level to all code (not in subroutines) "reachable" from the interrupt entrances (an instruction which occasionally skips assigns its level to the next two locations, a transfer assigns its level to its effective address(es), etc.). A subroutine call propagates its level to all of its possible exits.

With all of the main line code dealt with, we next examine each subroutine to see if all of its calls have had their hardware levels assigned. If so, we assign the minimum of the calling hardware levels as the hardware level of the entry to the subroutine. We then assign hardware levels to the rest of the code in the subroutine, and then loop back to find other subroutines all of whose calls have had their levels assigned.

This procedure should result in every line of executable code having a hardware level assigned to it. If some code has remained unassigned, then it is not ever executed. If there has been an attempt to assign to some code several hardware levels, there is probably a bug in the control structure.

Next we assign hardware levels to the data and variables of the system. First, all of the read-only variables (i.e., constants) are located and marked as such. All other variables are assigned the hardware level of the minimum of their referents. This should assign a hardware level to all variables, and every used word in the program, both code and data, will now have a hardware level assigned to it.

Now we assign software levels to the code. Each interrupt entrance is assigned a software level equal to its hardware level. Then, for all instructions which are neither at the returns of subroutines nor instructions which affect the interrupt system (e.g., LOCK or UNLOCK) we propagate the software levels as we did the hardware levels. The level of the instruction following one which affects the interrupt system should simply reflect whatever was done to the interrupt system, and then propagation can continue. Subroutines are a little more complicated.

We initially assign to each subroutine entrance a level of zero and propagate that assignment through the subroutine to determine a tentative exiting software level for each exit from the subroutine. Then, as we encounter a subroutine call instruction (either in the main code or in another subroutine) we

compare the software level of the call and the tentative software level of the subroutine entrance. If the calling level is lower-numbered than the subroutine level, each place that the subroutine could exit to is assigned the software level of the associated exit. If the calling level is higher-numbered than the subroutine level, the subroutine entrance is reassigned with the level of the call and the new level is propagated through the subroutine (which may entail changes in other subroutines). If the software level of any exit from the subroutine is changed, then every call to that subroutine thus far processed must have the affected exit reassigned and this new level must be propagated to the succeeding instructions. Since such a change can only increase an instruction's software level number (and there are only a finite number of levels), this procedure will eventually terminate. When it does terminate, each subroutine will have been assigned the maximum of the software levels of all of its calls, and the effects of such an assignment will have been propagated back through the calling sequences.

At this point, all executable code should have both hardware and software levels assigned to it. If, at any point, the system has attempted to assign two different software levels to a word (except as a result of reassigning a subroutine), it assigns the maximum. If, at any point, it has attempted to assign a software level larger than the hardware level, then there is a bug.

Now we are ready to verify that all of the data references are bug-free. First, many types of data references are "interrupt-proof"; e.g., an isolated LOAD or STORE reference. "Dangerous" references to variables must be checked, however. For each such reference, the scope of the reference must be identified. Within each such scope, determine the largest software level occurring anywhere within it (not forgetting to trace through any subroutines called). Then verify that the hardware level assigned to it is equal to the derived reference-software-level. Any violation is an error: if the hardware level is less than the software level, then the reference is not sufficiently locked to insure a bug-free reference; if the hardware level is greater than the software level, then the reference is excessively locked and perhaps system performance can be improved by increasing the software level of the reference.

This procedure should verify that there are no interrupt bugs in the control or data structures of a program. The ambiguities (e.g., in determining "dangerous" versus "safe" references) can usually be eliminated by the cooperation of the programmer; for example, the programmer could just use the op-code SSTORE (which would assemble in the same way as a STORE) to indicate a "safe" store, or he could use a different mnemonic

for potentially skipping instructions which are known, in the context, never or always to skip.

2.4 Summary

Our approach has been threefold: 1) we have analyzed exactly what properties and constructs within a system could give rise to multiprogramming conflicts; 2) we have written macros for our assembler which note the various protection structures and log any deviations (i.e., protections where none are needed, and portions which require additional protection); and 3) we have designed a groundwork from which a more complete and more automatic system could be built.

Finally, we have extended the theoretical basis of our system so that other types of multiprogramming structures, e.g., multiprocessing and polling systems, can be handled. We have designed the basis of an automatic programming system which would allow the writing of multiprogramming conflict-free programs with minimal unnecessary overhead. The rules, although difficult to incorporate into an assembly language, could easily be included in an appropriate higher level language.

We described this work at the IFIP/TC-2 Working Conference on Software for Minicomputers at Lake Balaton, Hungary; and a

written description of the work is to appear in a book published by North-Holland.

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author)

Bolt Beranek and Newman Inc,
50 Moulton Street
Cambridge, MA 02138

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

3 REPORT TITLE

QUARTERLY TECHNICAL REPORT NO, 3 INTERFACE MESSAGE PROCESSORS

4 DESCRIPTIVE NOTES (Type of report and, inclusive dates)

1 July 1975 to 30 September 1975

5 AUTHOR(S) (First name, middle initial, last name)

Bolt Beranek and Newman Inc,

6 REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
October 1975	27	

8a CONTRACT OR GRANT NO	9a. ORIGINATOR'S REPORT NUMBER(S)
F08606-75-C-0032	
b. PROJECT NO	Report No, 3182
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	

10 DISTRIBUTION STATEMENT

Distribution unlimited

11. SUPPLEMENTARY NOTES	12 SPONSORING MILITARY ACTIVITY

13 ABSTRACT

The ARPA computer network is a packet-switching store-and-forward communications system designed for use by computers and computer terminals. This Quarterly Technical Report briefly describes various aspects of network operation and maintenance (installation of IMPs, security activities, Network Control Center program modifications, network hardware maintenance, and developments related to the Terminal IMP, 316 IMP, Pluribus IMP, and Satellite IMP) and presents in detail a systematic method for detecting and preventing "interrupt" or "multi-programming" bugs.

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6811

UNCLASSIFIED

Security Classification

A-31409

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computers and Communication						
Store and Forward Communication						
ARPA Computer Network						
Packets						
Packet-switching						
Interface Message Processor						
IMP						
Terminal IMP						
TIP						
Pluribus						
Satellite IMP						
Access Control						
Accounting						
Private Line Interface						
PLI						
Broadcast Communications						
Acknowledgment						
Retransmission						
NCC						
Multiprogramming						
Interrupts						
Program Verification						